Improving Reuse of Attribute-Based Access Control Policies Using Policy Templates

Maarten Decat, Jasper Moeys, Bert Lagaisse, and Wouter Joosen

iMinds-DistriNet, KU Leuven, 3001 Leuven, Belgium
{first.last}@cs.kuleuven.be

Abstract. Access control is key to limiting the actions of users in an application and attribute-based policy languages such as XACML allow to express a wide range of access rules. As these policy languages become more widely used, policies grow both in size and complexity. Modularity and reuse are key to specifying and managing such policies effectively. Ideally, complex or domain-specific policy patterns are defined once and afterwards instantiated by security experts in their application-specific policies. However, current policy languages such as XACML provide only limited features for modularity and reuse. To address this issue, we introduce policy templates as part of a novel attribute-based policy language called STAPL. Policy templates are policies containing unbound variables that can be specified when instantiating the template in another policy later on. STAPL supports four types of policy templates with increasing complexity and expressiveness. This paper illustrates how these policy templates can be used to define reusable policy patterns and validates that policy templates are an effective means to simplify the specification of large and complex attribute-based policies.

Keywords: Access control, access control policies, attribute-based access control, reuse, modularity, policy templates.

1 Introduction

Access control is an important part of application-level security that constrains the *actions* of authenticated *subjects* on the *resources* in the application by enforcing *access rules*. Traditionally, access control was tightly coupled with the application code, making both hard to maintain. To address this, policy-based access control separates the access rules from the application code into declarative *access control policies* [15]. This approach improves modifiability by allowing the access rules to vary without having to change the application code. This approach also benefits separation of concerns by allowing application developers to focus on writing application code and security experts on specifying the access control policies in separate software modules.

However, while policy-based access control facilitates separation of concerns between application logic and access control logic, the challenge now is to effectively specify and manage the access control policies themselves. For example,

F. Piessens et al. (Eds.): ESSoS 2015, LNCS 8978, pp. 196–210, 2015.

```
When a subject tries to view the detailed status of a patient:
1. Deny if the subject is not a nurse.
2. Deny if the owning patient has withdrawn consent for the
   subject, unless the status of the patient is critical or
   if the subject has triggered breaking-the-glass, which
   should be logged.
3. Deny if the subject is not on shift.
4. Permit if the subject is currently treating the owning
   patient.
5. Permit if the subject is of the emergency department.
6. Deny otherwise.
```

Listing 1.1. A running example of six access control rules from an industrial e-health case study [6]. The application is a patient monitoring system and the rules are specified by the hospital that employs the system. The first rule for which the condition holds should provide the overall decision.

take the access rules of Listing 1.1. These rules are taken from an industrial e-health case study, i.e., a system provided to hospitals for monitoring patients at their homes [6]. These six rules are only an excerpt of the complete set of rules, but already require the concepts of ownership, patient consent, the patient status, breaking-the-glass procedures, the departments of the hospital and the treating relationship between physicians and patients. Moreover, these rules should be combined correctly, in this case being that the first rule for which the condition holds should provide the overall decision. As a result, current applications require a policy language in which a wide spectrum of rules can easily be expressed and combined into one unambiguous composite policy.

Of the current policy languages, XACML [1] partially achieves this. Firstly, by employing attribute-based access control (ABAC, [11]), XACML supports most of the rules required by the case study. Secondly, by employing policy trees (see Figure 1), XACML supports structuring multiple rules and reasoning about how these relate in case of conflict. However, XACML does not allow to specify and manage large policies *effectively*. In the example rules of Listing 1.1, the roles of a subject are a well-known access control concept [7], the concept of patient consent is specific to the domain of e-health, the status of a patient is specific to the patient monitoring system and the departments are specific to the hospital. Ideally, these concepts are defined once as patterns that can be reused within their respective domains. Moreover, each pattern is ideally defined by its respective expert, i.e., an access control expert, an e-health domain expert, the application provider or an expert of the hospital. In terms of attribute-based policies, such patterns would consist of rules and the definitions of the attributes required by these rules. However, XACML does not support attribute definitions and only allows to include other policies without modification such that slight variations cannot be modularized in a single pattern.

To address these limitations, this paper introduces *policy templates* in attribute-based tree-structured policies as part of a novel policy language called

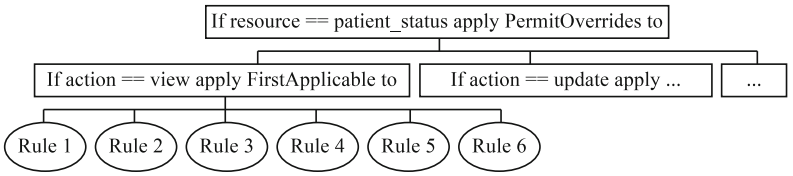

Fig. 1. Example of a policy tree that combines the six rules of the introduction. The leafs of the tree are rules. The intermediate nodes are policies that specify a target and the combination algorithm to combine the results of their children.

the Simple Tree-structured Attribute-based Policy Language or STAPL. Policy templates are policies containing unbound variables that can be bound to values, expressions, rules or even other policies when instantiating the template later on. In addition, STAPL allows to package these templates in reusable *policy modules*. More precisely, STAPL supports four types of policy templates of increasing expressiveness:

1. *simple policy references*, which include other policies without modification,
2. *simple policy templates*, which include variations of policies by extending policy references with unbound variables,
3. *modules of policy templates with attribute definitions*, which encapsulate policy templates and the definitions of the attributes they require,
4. *modules of policy templates with specialized types of attributes*, which extend STAPL with specialized attributes and functions that reason about them.

This paper illustrates how these policy templates and modules can be used to modularize access control patterns and thereby increase policy comprehensibility, facilitate policy reuse and facilitate separation of concerns between the different stakeholders in policy specification.

The rest of this paper is structured as follows. Section 2 further illustrates the problem statement of this work. Section 3 describes STAPL and the different types of policy templates it supports. Section 4 validates the potential of policy templates. Section 5 discusses related work and puts the results in a broader perspective. Section 6 concludes this paper.

2 Context and Problem Illustration

Policy-based access control is an approach to access control in which the access rules are specified in declarative policies that are enforced and evaluated by specialized components in the application. As such, the access rules can vary and evolve separately from the application code and both can be specified by their respective experts, a principle called separation of concerns [14].

Over the last decades, multiple models have been proposed to reason about access control and specify access control policies. The state of the art supports a wide range of complex rules by combining two technologies: attribute-based

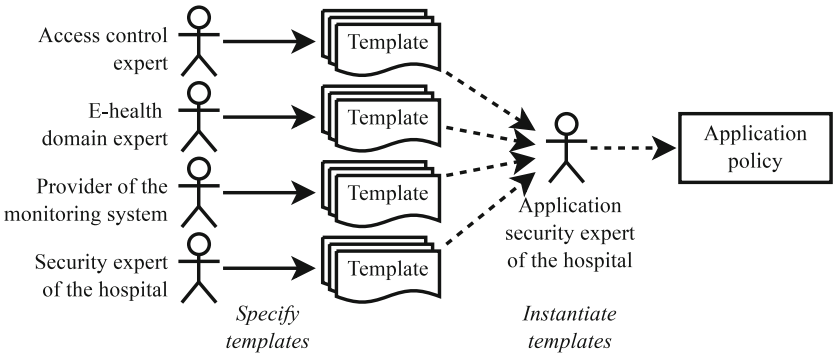

Fig. 2. Our vision on policy specification: an application security expert should be able to specify policies for his or her specific organization and application by instantiating policy patterns represented as templates pre-defined by other experts

access control (ABAC, [11]) and policy trees. Firstly, ABAC is a recent access control model that expresses rules in terms of key-value properties of the subject, the resource and the environment called attributes. Examples of such attributes are the subject roles, the resource location and the time. As such, ABAC supports rules such as identity-based permissions, roles, ownership, time, location, consent and breaking-the-glass procedures. Secondly, the most widely-used language for attribute-based policies XACML [1] additionally allows to combine multiple attribute-based rules in a single policy as a tree, a concept also present in the literature (e.g., [4,12]). As illustrated in Figure 1, each element in such a policy tree defines to which requests it applies and how the results of its children should be combined, e.g., a permit overrides a deny. As said, these technologies together support a wide range of rules. However, specifying complex rules such as the patient consent rule (Rule 2 of Listing 1.1) using these technologies is not trivial. Consequently, managing large sets of such rules is even more challenging.

Modularization and reuse are key to managing this complexity effectively. To illustrate this, again take the example rules from Listing 1.1. Parts of these rules are well-known access control concepts such as the roles of a subject, while other parts are specific to the domain of e-health, specific to the application or specific to the hospital. Ideally, these parts are specified once by their respective experts as patterns that can be reused in multiple policies of their respective domains:

- An *access control expert* defines patterns for well-known access control concepts, such as ownership and hierarchical roles. These patterns can then be reused in any access control policy.
- An *e-health domain expert* defines patterns for frequent e-health rules, such as patient consent and the breaking-the-glass procedure in the example rules. These patterns can then be reused in any e-health policy.
- An *application provider* defines patterns for rules and attributes specific to its application, such as the status of a patient in the example rules. These patterns can be reused in any policy that applies to this application.

– A *security expert of the hospital* defines patterns for hospital-specific rules and attributes, such as the shifts of nurses and the departments of the hospital in the example rules. These patterns can be reused in policies for any application employed by the hospital.

After these patterns have been defined, the *application security expert of the hospital* can specify the policy for this hospital and his or her specific application by simply instantiating these patterns. This vision is illustrated in Figure 2.

However, the state of the art with XACML does not support such patterns. More precisely, the patterns in the examples above consist of five types of definitions: (1) rules that can be included in other policies without modification, e.g., the rule that checks patient consent, (2) rules that can be included in other policies with slight modifications, e.g., the rule that checks whether the subject is on shift, in which the shift hours should be specified by the hospital, (3) attributes that are required by these rules, e.g., the owner of a resource or the current time, (4) attributes that can be used by other rules later on, e.g., the roles of a subject or the status of a patient, and (5) the possible values of a certain attribute, e.g, the departments of the hospital and the roles of the employees in the hospital. Of these five types, XACML only allows the first, i.e., to include other policies literally by using policy references. The more advanced ALFA language for generating XACML policies [8] does support attribute definitions as well, but is still limited to including policies without modification.

To address these limitations, this paper introduces policy templates in attribute-based tree-structured policies as part of a novel policy language called STAPL. Policy templates are partially specified policies that contain unbound variables that can be specified later on. STAPL additionally supports encapsulating these policy templates with the definitions of the attributes they require into self-contained policy modules. Policy templates have been put forward in formal work about policies trees [2] and were part of the Ponder policy language [5], both more than a decade ago. However, policy templates have not been applied in state-of-the-art policy languages of which the complexity actually increases the need for them.

3 Policy Templates in STAPL

This paper introduces policy templates in attribute-based tree-structured policies as part of a policy language called STAPL[1]. More precisely, STAPL supports (1) simple policy references, (2) simple policy templates, (3) modules of policy templates with the definitions of the attributes they require and (4) the extension of the previous with specialized types of attributes. This section first introduces STAPL and then discusses each of these.

STAPL Basics. STAPL is a policy language designed to easily specify attribute-based tree-structured policies. In other words, STAPL takes on a policy model similar to XACML, but is designed for easier specification (amongst

[1] The code of STAPL publicly is available at `https://github.com/stapl-dsl/`

```
action.id = SimpleAttribute(String)
resource.owner = SimpleAttribute(String)
subject.treating = ListAttribute(String)
subject.roles = ListAttribute(String)
environment.now = SimpleAttribute(Time)
...
Policy := when (action.id == "view" and resource.type ==
  "patient_status") apply FirstApplicable to (
    Rule := deny iff (not "nurse" in subject.roles),
    Policy := apply PermitOverrides to (
      Rule := deny iff (subject.id in
        resource.owner_withdrawn_consents),
      Rule := permit iff (resource.patient_status == "bad"),
      Rule := permit iff (subject.triggered_breaking_glass)
        performing (log(subject.id + " broke the glass"))
    ),
    Rule := deny iff (not (environment.now >= 08:00 and
      environment.now <= 17:00)),
    Rule := permit iff (resource.owner in subject.treating),
    Rule := permit iff (subject.department == "emergency"),
    Rule := deny
)
```

Listing 1.2. The STAPL definition of the example rules of Listing 1.1 without the use of policy templates

```
// example definitions
def defaultDeny = Rule := deny
def denyIfNotOnShift = Rule := deny iff (
  not (environment.now >= 08:00 and environment.now <= 17:00))
def permitIfTreating = Rule := permit iff (
  resource.owner in subject.treating)
// example usage
action.id = SimpleAttribute(String)
...
Policy := when (...) apply FirstApplicable to (
  ...
  denyIfNotOnShift,
  permitIfTreating,
  ...
  defaultDeny
)
```

Listing 1.3. Example definitions and usage of policy references

others). Listing 1.2 shows the STAPL specification of the example rules of Listing 1.1. As shown, the attributes to be used in the policy are defined first (lines 1-6). These attributes have a type and can be single-valued or multi-valued. Then the policy is defined in terms of these attributes (lines 7-22). STAPL employs policy trees consisting of Policies and Rules: Rules are the basic elements of a STAPL policy and Policies are collections of multiple Rules or other Policies. Every Policy specifies to which requests it applies by means of an attribute-based target (following the keyword `when`, see lines 7-8) and specifies how to combine the results of its children by means of a combination algorithm such as `PermitOverrides` or `FirstApplicable`. STAPL is defined as an internal domain-specific language (DSL) in Scala because of its powerful features for both DSLs and modularity. Of the different features of STAPL, this paper only focuses on the policy templates, which are explained next.

Simple Policy References. Policy references are the simplest type of policy templates provided by STAPL. Similar to XACML, policy references allow policies to be reused without modification and do not include attribute definitions. Listing 1.3 illustrates how to define a policy reference (lines 2-6) and use one (lines 10-16). As shown, policy references are Scala functions (denoted by the keyword `def`) that do not require arguments and return a STAPL Rule or Policy.

Simple Policy Templates. Policy templates extend policy references with unbound variables. These unbound variables can be literal values, attribute references, attribute-based expressions or even other Rules or Policies. Listing 1.4 illustrates how to define a policy template (lines 2-9) and use one (lines 11-18). The first example generalizes `denyIfNotOnShift` of Listing 1.3, the second encapsulates the pattern of permitting in a certain condition and denying otherwise. As shown, policy templates are Scala functions that require arguments and return a STAPL Rule or Policy.

Modules of Policy Templates with Attribute Definitions. Thirdly, STAPL supports encapsulating policy templates with the definitions of the attributes required by these templates. This decreases the chance for incorrect attribute definitions and fully encapsulates a policy pattern as a reusable and self-contained module. Listing 1.5 illustrates how to define such a policy module (lines 2-12). As shown, a policy module is a Scala trait that extends the trait `BasicPolicy`. A Scala trait is similar to a class, but allows multiple inheritance. The trait `BasicPolicy` defines the variables `subject`, `resource`, `action` and `environment` so that policy modules can assign attributes to them. Listing 1.5 also illustrates how to import the defined modules, i.e., by extending the scope in which the policies are defined using the Scala traits (line 14). Since Scala allows multiple inheritance using traits, the scope can be extended with any number of required modules. Moreover, since Scala allows traits to extend other traits, policy modules can extend existing modules as well. Amongst others, this can be used to express that a certain module depends on other modules, as illustrated by the `Treating` module (line 9).

```
// example definitions
def denyIfNotOnShift(start: Time, stop: Time) =
  Rule := deny iff (not (environment.now >= start
    and environment.now <= stop))
def denyUnless(condition: Expr) =
  Policy := apply PermitOverrides to (
    Rule := permit iff (condition),
    defaultDeny // see Listing 1.3
  )
// example usage
action.id = SimpleAttribute(String)
...
Policy := when (...) apply PermitOverrides to (
  ...
  denyIfNotOnShift(08:00, 17:00),
  ...
  denyUnless(subject.department == "emergency")
)
```

Listing 1.4. Example definitions and usage of policy templates

```
// example definitions
trait Shifts extends BasicPolicy {
  environment.now = SimpleAttribute(Time)
  def denyIfNotOnShift(...) = ... // see Listing 1.4
}
trait Ownership extends BasicPolicy {
  resource.owner = SimpleAttribute(String)
}
trait Treating extends Ownership {
  subject.treating = ListAttribute(String)
  def permitIfTreating = ... // see Listing 1.3
}
// example usage
object example extends BasicPolicy with Shifts with Treating {
  // notice: no attribute definitions here
  Policy := when (...) apply PermitOverrides to (
    ...
    denyIfNotOnShift(08:00, 17:00),
    permitIfTreating,
    ...
  )
}
```

Listing 1.5. Example definitions and usage of policy modules that contain both policy templates as well as the definitions of the attributes they require

```
1  ... // definition of the role attribute omitted
2  // example definition of the role hierarchy
3  val employee = new Role()
4  val nurse = new Role(employee)
5  val headNurse = new Role(nurse)
6  ...
7  // example usage
8  Policy := when (...) apply PermitOverrides to (
9    // this applies to all types of nurses
10   Rule := deny iff (not subject.hasRole(nurse))
11 )
```

Listing 1.6. Example usage of a specialized attribute type for hierarchical roles. The definition of this attribute type required 35 lines of code and is omitted because of space limitations. Notice that `subject` has been extended with a specialized function to reason about hierarchical roles.

Policy Templates with Specialized Attributes. Finally, policy modules can extend the core functionality of STAPL (i.e., single-valued or multi-valued attributes of simple types such as numbers, strings, booleans or dates, and simple operators such as `==`, `in` and `<=`) with specialized types of attributes and functions that reason about these attributes. This functionality can be used to express for example hierarchical roles. In this case, the application security expert would like to define the hierarchy of roles and test whether a subject owns a certain role in this hierarchy. Listing 1.6 shows an example of how such an attribute could be used. For space reasons, Listing 1.6 omitted the definition of this specialized attribute, which consists of the definitions of the attribute itself, the mapping of these attributes to STAPL expressions and the extensions to `subject`, `resource`, `action` and `environment` (35 lines of code in total). This shows that policy modules with specialized attributes are the most expressive type of policy patterns offered by STAPL, but also the most complex.

4 Validation

The previous section discussed the different types of policy templates supported by STAPL. This section validates whether they can be used to create reusable policy modules and that these modules can be used to separate the different stakeholders involved in policy specification.

4.1 Approach

To validate the potential of policy templates, we applied them to an existing policy by consistently factoring out rules and attribute definitions into policy modules. Afterwards, we validated whether each resulting policy module falls within the expertise of a single stakeholder and that each module can be reused within its respective domain. If so, this shows that policy templates can separate

the different stakeholders in policy specification, enable reuse and that realistic policies can be specified by instantiating such templates. Of course, this process is mainly meant for validation purposes and STAPL aims for a situation in which policies are specified using policy templates from the start.

The original policy applies to viewing the status of a patient monitored by a patient monitoring system. This policy was specified by the authors before the work on STAPL as part of the e-health case study that was also employed in the running example of Listing 1.1 [6]. The different stakeholders in this policy are the five stakeholders identified before: the general access control expert, the e-health domain expert, the application provider, the security expert of the hospital and the application security expert of the hospital.

The policy was originally specified in XACML and was first translated to STAPL literally to achieve a fair evaluation[2]. In terms of STAPL, the policy consists of 23 Rules (plus 7 default denies) divided over 12 Policies, resulting in a policy tree of depth 5. The policy requires 30 different attributes, of which 17 subject attributes, 11 resource attributes, 1 action attribute and 1 environment attribute.

4.2 Modularization Results

The result of the modularization of the policy is shown in Figure 3. We can make several interesting observations from this figure.

Observation 1: Separation of concerns. Firstly, Figure 3 shows that it is possible to apply policy templates so that every policy module can be specified by exactly one expert. As a result, the different roles of the different experts outlined above can be separated appropriately:

1. The access control expert specifies the general policy modules such as `Roles`, `Time`, `Ownership`, `Location` and `GeneralTemplates`. `GeneralTemplates` amongst others contains the definitions of `defaultDeny` and `denyUnless()` shown in Listing 1.3 and Listing 1.4. `Roles` defines the specialized attribute for hierarchical roles.
2. The domain expert specifies the policy modules that apply to the domain of e-health in general, which in this case is `Consent` and `BreakingGlass`.
3. The application provider specifies the policy modules that are specific to the application, in this case mainly `PatientMonitoringSystem`, which defines the resource attributes supported by the application. Of these attributes, `resource.patient_status` represents a gradation of patient statuses. This specialized attribute has been generalized into the separate module `PatientStatus` that can be reused by the application provider later on.
4. The security expert of the hospital defines the policy modules that are specific to the hospital. These comprise the role hierarchy of the employees of the hospital, its different departments, the definitions of 13 attributes that

[2] The original policy and the result of the refactoring are available at `https://distrinet.cs.kuleuven.be/software/stapl/`

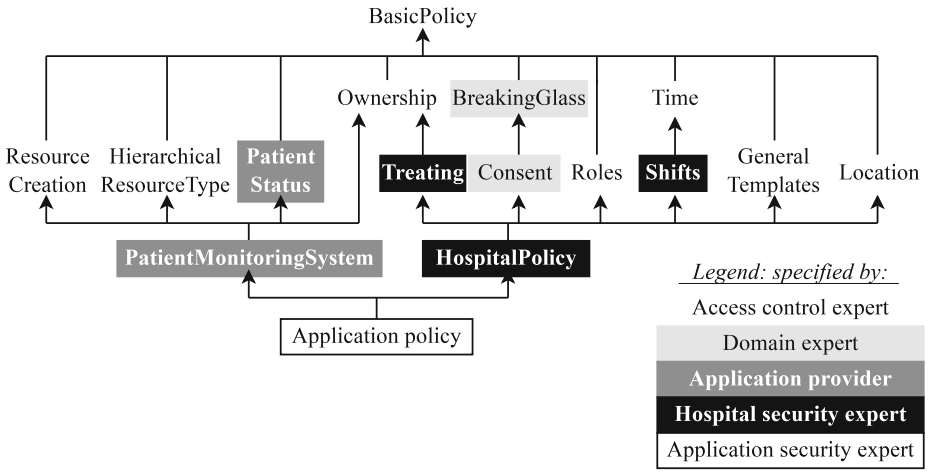

Fig. 3. The hierarchy of policy modules resulting from refactoring the original policy of the e-health case study. Arrows represent a dependency between policy templates and the colors indicate which policy template fits the expertise of which stakeholder.

apply to its employees (e.g., `subject.department` and `subject.is_discharged`) and the rules to reason about shifts and the treating relationship between medical personnel and patients. Again, the latter have been generalized into separate modules that can be reused within the hospital.

5. Finally, the application security expert only specifies the specific policy that holds for the patient monitoring system as used by the hospital. There was no need to define other attributes, i.e., all attributes used by this policy were pre-defined in policy modules.

Observation 2: Reusability. As a second observation, the modules defined by the different experts indeed apply to their own domain and can thus be reused in these domains. For example, the templates in the modules defined by the application provider can be reused by any organization using its application and the templates in the modules defined by the access control expert can be reused in other access control policies regardless of the application or the domain.

Observation 3: Number of policy modules. Thirdly, Figure 3 shows that it was possible to extract no less than 14 modules from a policy consisting of 23 rules. This shows how common reusable patterns are in access control policies and indicates the potential of policy templates for simplifying policy specification. However, we do not expect the number of modules that can be extracted from a larger policy to grow linearly with the number of rules in that policy. On the contrary, given the reusability of the extracted modules discussed in the previous observation, we expect the number of defined modules to stagnate, but the reuse of the templates in these modules to grow.

Observation 4: Fine-Grained Module Hierarchy. Finally, it is worth noting that most of the 14 modules only define a small number of templates and attributes. By employing module dependencies, these fine-grained templates can then be structured in a hierarchy of modules of gradually increasing size in order to only include the specific features needed in the final policy.

4.3 Analysis of the Modules

In addition to the resulting module hierarchy, it is also interesting to analyze the different types of templates used in this hierarchy. More precisely, the 14 modules of Figure 3 define 2 policy references (`defaultDeny` and `defaultPermit`), 6 policy templates (e.g., `DenyUnless()` and `denyIfNotOnShift()`), 30 definitions of standard attributes, 3 definitions of specialized attributes (roles, hierarchical resource types and the status of a patient) and 3 functions for using these specialized attributes. The final policy did not define any other attributes and instantiated 0 policy references, 9 policy templates and 9 functions for using the specialized attributes[3]. These numbers show four interesting observations:

Observation 1: references vs templates. Apart from `defaultDeny` and `defaultPermit`, all other templates required unbound variables in order to be reusable. This validates the original claim that policy references by themselves are not powerful enough to provide reusability.

Observation 2: template definition vs instantiation. The number of template instantiations is significantly higher than the number of template definitions. This is mainly due to the use of roles (7x) and `DenyUnless()` (6x) throughout the policy. Both patterns frequently occurred in the policy, while the other templates are more specialized and were used only once. In the former case, modularization provides the benefit of specifying a frequent pattern only once; in the latter case the benefit is having other experts specify complex rules and instantiating these using a simple interface.

Observation 3: no attribute definitions in the application policy. Finally, we observe that all attributes required by the final application policy are defined in the modules and that the policy did not define any attributes itself. This fits reality well: a policy cannot simply define a new attribute since the available resource attributes are determined by the application and defining a new subject attribute requires assigning values for this attribute to the subjects of the organization. In fact, policy modules provide a means to actually consolidate these definitions in both an application-specific or organization-specific module.

4.4 Effect of Templates on the Policy Size

Finally, Table 1 provides metrics about the effect of policy templates on the size of a policy. Table 1 first compares the number of lines of code required to express

[3] The final policy did not employ any policy references because `defaultDeny` and `defaultPermit` were encapsulated in `DenyUnless()` and `PermitUnless()`.

Table 1. Metrics about the size of the policy without and with policy templates

	Without	With
#Lines of code for policy definition	57	56 (98.2%)
#Statements in the policy definition	367	205 (55.6%)

the policy (not counting attribute definitions). As shown, the use of policy templates does not significantly affect this number. However, as shown next in the table, the use of policy templates does significantly lower the number of *statements* required to express the policy (e.g., `Policy`, `when`, `==`). This is the result of replacing complex rules or frequently occurring patterns with templates that require less statements to instantiate, such as the consent rule or `DenyUnless()` respectively. While these numbers only comprise one case study, they do show that policy templates have the ability to simplify policy specification significantly.

5 Related Work and Discussion

In the previous sections, we introduced and validated policy templates in attribute-based tree-structured policy languages as a means to improve reuse and modularity. To the best of our knowledge, this work is the first to discuss policy templates in the domain of attribute-based access control. Before this work, policy templates have also been discussed by Wies et al. [17], Casassa Mont et al. [3], Bonatti et al. [2] and as part of the Ponder policy language [5]. Similar to our vision, Wies et al. and Casassa Mont et al. discuss policy templates as part of a larger view on policy management in which templates are a means to allow different experts to cooperate. Bonatti et al. on the other hand discuss policy templates as part of their formal work on modular policy composition using policy trees. As a result, our definition of policy templates aligns to theirs, but fits in the vision of Wies and Casassa Mont. Finally, Ponder is an extensive policy language that amongst others also supports policies with unbound variables, but did not yet employ attribute-based access control or policy trees. The main difference between this work and these four is our focus on attribute-based tree-structured policy templates supported by a concrete policy language, the generalization of policy templates into policy modules and the validation of the potential of policy templates. In addition to these authors, templates have also been discussed in the evolution from role-based access control (RBAC, [7]) to ABAC (e.g., [9,13]). Such role templates were eventually generalized into attribute-based policies, for which STAPL in turn introduces policy templates.

This work started from the observation that state-of-the-art attribute-based policy languages can express a wide range of rules, but that policy specification has also become more complex. This observation aligns to recent visions of amongst others Sandhu [16]. Sandhu argues that attribute-based access control can offer numerous benefits, but also provides risks and challenges such as the increased complexity of attribute management. Similarly, NIST recently published

a guide to ABAC [10] containing open challenges and a vision on the multiple roles involved in employing ABAC in an enterprise. This work fits into both visions with a specific focus on efficient policy management and specification. However, when scoping this work in the visions of Sandhu and NIST, it also becomes clear that policy templates are only a first step to an effective deployment of ABAC. For example, while STAPL allows to simply import an attribute definition, the largest effort will be in providing values for this attribute for the appropriate subjects or resources. Therefore, the next step for this work is to extend modular policy management with modular attribute management.

An interesting side-effect of our work on policy modularity is the possibility to explicitly separate the roles of the provider of an application and the organization using it (see Figure 3). In terms of attribute-based policies, the application provider determines the structure of the resources in the application, i.e., the types of resources, their attributes and the actions they support. The organization on the other hand determines the structure of the subjects, i.e., the types of subjects and their attributes. The policies then combine both by expressing which subjects can perform which actions on which resources. As shown in our validation, policy templates provide a means to concretize the structure of the subjects and the resources, which can prove valuable to simplify the correct specification of access rules, for example by employing this information for completeness checking. Towards the future, we plan to further investigate this application of policy templates.

Finally, we want to mention that STAPL is currently built as a Scala DSL. Scala provides powerful features for both DSLs and modularity and allows STAPL to be directly employed in Scala or Java applications. However, in the long run, it is our aspiration that the concepts presented in this paper are incorporated in language-independent policy languages, such as XACML itself.

6 Conclusion

In this paper, we introduced policy templates as a means to improve reuse in attribute-based tree-structured policy languages such as XACML. Our policy language STAPL supports four types of policy templates ranging from simple policy references to modules encapsulating policy templates and the specialized attributes required by these templates. This paper showed that these policy templates can be used to set up a hierarchy of fine-grained reusable policy modules. Each such module can encapsulate a policy pattern, can be defined by the appropriate expert and can be reused within the domain of that expert. As such, it is our aspiration that STAPL is a first step towards a policy specification process in which policy modules are defined once by experts and access control policies are *composed* by instantiating these modules.

References

1. eXtensible Access Control Markup Language (XACML) Version 3.0. OASIS Standard (2013)
2. Bonatti, P., De Capitani di Vimercati, S., Samarati, P.: An algebra for composing access control policies. ACM Trans. Inf. Syst. Secur. 5(1) (February 2002)
3. Casassa Mont, M., Baldwin, A., Goh, C.: Power prototype: towards integrated policy-based management. In: IEEE/IFIP Network Operations and Management Symposium (2000)
4. Crampton, J., Huth, M.: An authorization framework resilient to policy evaluation failures. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 472–487. Springer, Heidelberg (2010)
5. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The Ponder policy specification language. IEEE POLICY (2001)
6. Decat, M., Lagaisse, B., Joosen, W.: Middleware for efficient and confidentiality-aware federation of access control policies. Journal of Internet Services and Applications (2014)
7. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. TISSEC (2001)
8. Giambiagi, P., Rissanen, E., Nair, S.: Axiomatics Language for Authorization (ALFA). Announced to be Standardized as XACML Profile (April 2014)
9. Giuri, L., Iglio, P.: Role templates for content-based access control. ACM RBAC (1997)
10. Hu, V., Ferraiolo, D., Kuhn, R., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K.: Guide to Attribute Based Access Control (ABAC) Definition and Considerations. NIST Special Publication (2014)
11. Jin, X., Krishnan, R., Sandhu, R.: A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) DBSec 2012. LNCS, vol. 7371, pp. 41–55. Springer, Heidelberg (2012)
12. Li, N., Wang, Q., Qardaji, W., Bertino, E., Rao, P., Lobo, J., Lin, D.: Access control policy combining: Theory meets practice. ACM SACMAT (2009)
13. Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust-management framework. IEEE Security and Privacy (2002)
14. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM 15(12), 1053–1058 (1972)
15. Samarati, P., de Capitani di Vimercati, S.: Access control: Policies, models, and mechanisms. In: Focardi, R., Gorrieri, R. (eds.) FOSAD 2000. LNCS, vol. 2171, p. 137. Springer, Heidelberg (2001)
16. Sandhu, R.: The authorization leap from rights to attributes: Maturation or chaos? In: Proceedings of the 17th ACM Symposium on Access Control Models and Technologies, SACMAT 2012. ACM (2012)
17. Wies, R.: Using a classification of management policies for policy specification and policy transformation. In: Integrated Network Management IV, pp. 44–56. Springer (1995)